生成树机制实验报告

实验内容

完善stp.c中 static void stp_handle_config_packet(stp_t *stp,stp_port_t *p,

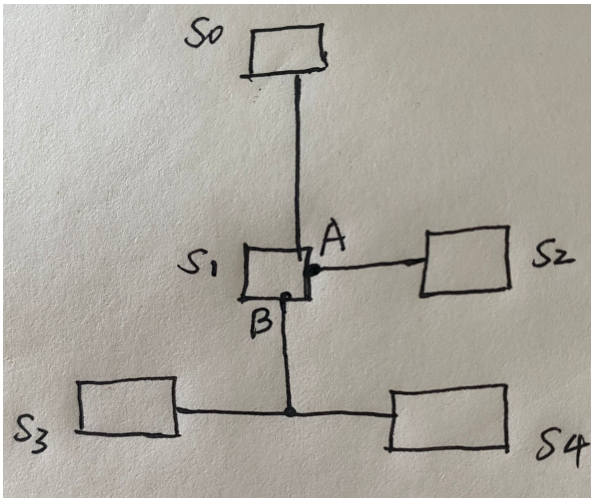
struct stp_config *config)

实验流程

1.各端口数据结构

最重要的三个数据结构是: stp_config，stp_port,和stp

首先关注同一个网段的概念，示意图如下：



由图1可以看出：若S0为根结点，这张图片中显示了三个网段(S0,S1)、 (S1,S2)、 (S1,S3,S4)为同一个网段。如果对于网段(S1,S2)而言，端口A是离根结点最近的端口（此时端口B甚至不属于网段(S1,S2)中，因此无法参与比较），同时对于网段(S1,S3,S4)而言，端口B是离根结点最近的端口，所以在各自的信息记录里，不同网段的Config消息和端口自身的消息是不同的。

S1结点的端口可以属于不同的网段。

```
struct stp_config{
    struct stp_header header; //定义好的头部格式
    u8 flags;                 // 本实验设置为0
    u64 root_id;              // 本条Config消息认为的根结点
    u32 root_path_cost;       // 记录到根结点的开销，这里到根结点的开销记录的是本网段内
    u64 switch_id;            // 发送Config消息的结点
    u16 port_id;              // 发送本Config消息的端口
    /*这里的Config消息是同一个网段内共享，对于同一个结点而言，不同的端口可能属于不同的网段，同时同一网段中也有不同的结点，这里的Config消息记录的就是本网段中，所有的结点当前都会存储：本网段优先级最高的结点和端口
    以上图为例，S1、S3、S4网段中，记录的Config消息中应该是：
    root_id = S0
    root_path_cost = 1 (因为根端口此时为B，而我们从整个网段的角度考虑，B到S0的路径长度为1)
    switch_id = S1 (不管是S3、S4还是S1),都是记录本网段的最高优先级的结点
    port_id = B (因为该网段所有的消息收发都是通过最上游的B依次向下转发)*/
    u16 msg_age;    // 在当前时刻STP包或者说Config消息的存活时间
    u16 max_age;    // 最大延时
    u16 hello_time; // 发送消息间隔
```

```
    u16 fwd_delay; //本实验中不考虑
}_attribute_((packed));
```

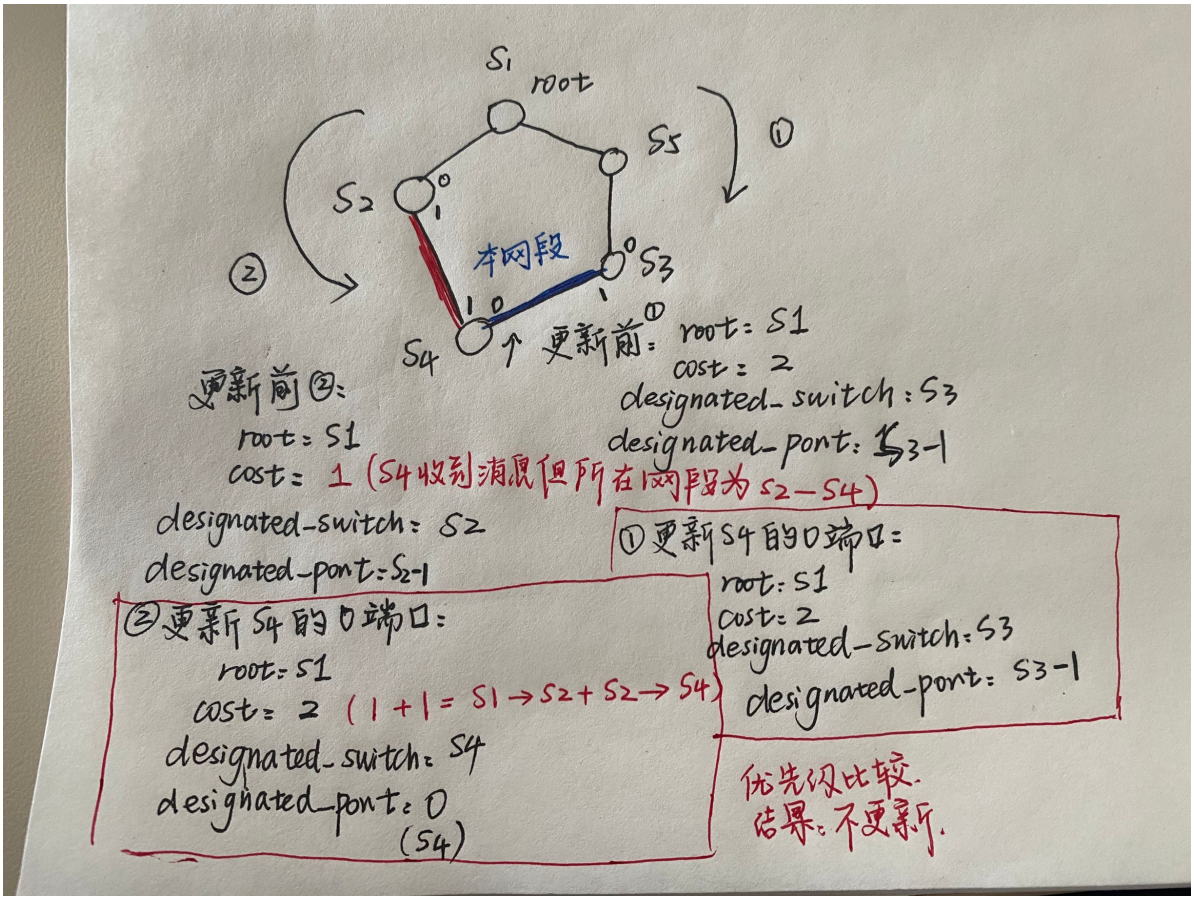
然后分析端口的信息，端口的信息其实也就是该端口的自身信息和存储在该端口的Config消息。这个Config消息与端口所在网段有关，是通过多轮的消息迭代与更新修改得到的。

```
struct stp_port{
    stp_t *stp;  // 指向端口所在的结点 stp

    int port_id; // 端口id号
    char *port_name;
    iface_info_t *iface;

    int path_cost;
    /* 过端口所在网段需要的花销，均为 1
    实际是因为，通常发送的Config消息来自上一跳，例如图1中 S3 和 S4 接收来自 S1 的消息，
    但 S1 的 Config 是自身到根结点的花销，所以 S3/S4 到根结点的花销 = S1 到根结点的花销
+   上一跳结点(S1)到自身(S3/S4)的一跳所经历的花销，这里均设为 1*/

    u64 designated_root;
    /*端口认为的根结点*/
    u64 designated_switch;
    /*上一跳结点，也就是（谁）把这个Config消息发送给我，以S3为例，就记录 S1*/
    int designated_port;
    /*上一跳端口,哪个端口把这个Config消息发送给我，以S3为例，就记录B */
    int designated_cost;
    /*网段内根端口到根结点的开销*/
};
```

接下来是stp的信息，实验函数中给出了 stp_port_t *p 和 stp_t *stp,p端口所在的结点为stp结点

```
struct stp{
    u64 switch_id;
    u64 designated_root;
    int root_path_cost;
    stp_port_t *root_port;
    // 认为的根结点，到根结点的路径开销，根端口
    long long int last_tick; // 计时器交换
    stp_timer_t hello_timer; // 消息发送的间隔时间

    int nports;
    stp_port_t ports[STP_MAX_PORTS];
    // 结点的端口列表
    pthread_mutex_t lock;
    pthread_t timer_thread;
    // 设置线程和锁
};
```

2. 从非指定端口到指定端口



这里主要解释结点的更新过程，是否需要更新为指定端口：

> 先构造上述一共 5 个结点的网络：根结点为 S1，我们先假设（1）号消息，从 S1 -> S5 -> S3
>
> 然后有(2) 号消息，从 S1 -> S2 -> S4。观察这两个消息对于(S4,S3)网段指定端口的更新情况。

1. (1)消息发送到 S3 的 端口1，我们此时知道

```
root = S1;
cost = 2;
// 因为对于网段(S3-S4),考虑根端口S3到根结点的距离，得到本网段到根结点的距离为 2
designated_switch = S3;
designated_port = S3 -eth1;
```

2. 由于(1)消息发送到 S3 的端口1，S3-eth1目前为指定端口，所以需要修改同一网段的S4-eth0的信息(此端口处的Config消息就是S3-eth1端口的Config消息)

```
root = S1;
cost = 2;
// 本网段到根结点的距离为 2
designated_switch = S3;
designated_port = S3 -eth1;
```

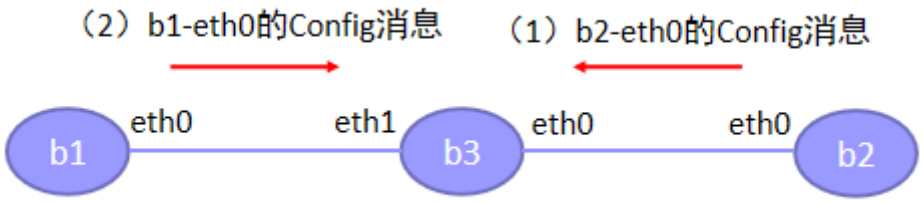
3.(2)消息发送到 S4 的eth0端口，这时考虑的网段是(S2,S4)。S4-eht0端口的Config消息为

```
root: S1
cost : 1
designated_switch: S2
designated_port: S2-eth1
```

4.由消息（2），更新S4的0 端口，我们先假设如果修改了，Config消息应该为:

```
root: S1
cost: 2 // S1-> S2 + S2 -> S4
designated_switch: S4  /*因为只有S4才属于(S4,S3)网段，这个信息可以由本地的stp->switch_id 得到 */
designated_port: S4-0 /*更新S4结点时更新自身的端口*/
```

5.实际上4.的消息并没有被写入，需要与2.进行比较，由于第三项上一跳结点的ID 2.中记录的值为S3,所以2 的优先级更高，4.不发生修改。



再说明这一个例子的情况。

首先b3-eth0收到了 b2-eth0 发送的Config消息，因为结点ID b2 < b3，所以 b3-eth0认为 b2才是根结点，而 b3-eth0 的ID 在 (b2,b3)网段中优先级低，所以在（1）消息情况下，变为非指定端口。

然后收到了（2）消息，修改了b3-eth1 的Config信息，此时认为的根结点是 b1 ,并且b3-eth0 记录的根节点的信息也要进行修改，root_path_cost = 0 + 1 = 1，此时的优先级比本地存储的b2 的优先级要高，所以对于(b3-b2)这一网段，指定端口由 b2-eth0 变为 b3-eth0。

3.初步的实验代码

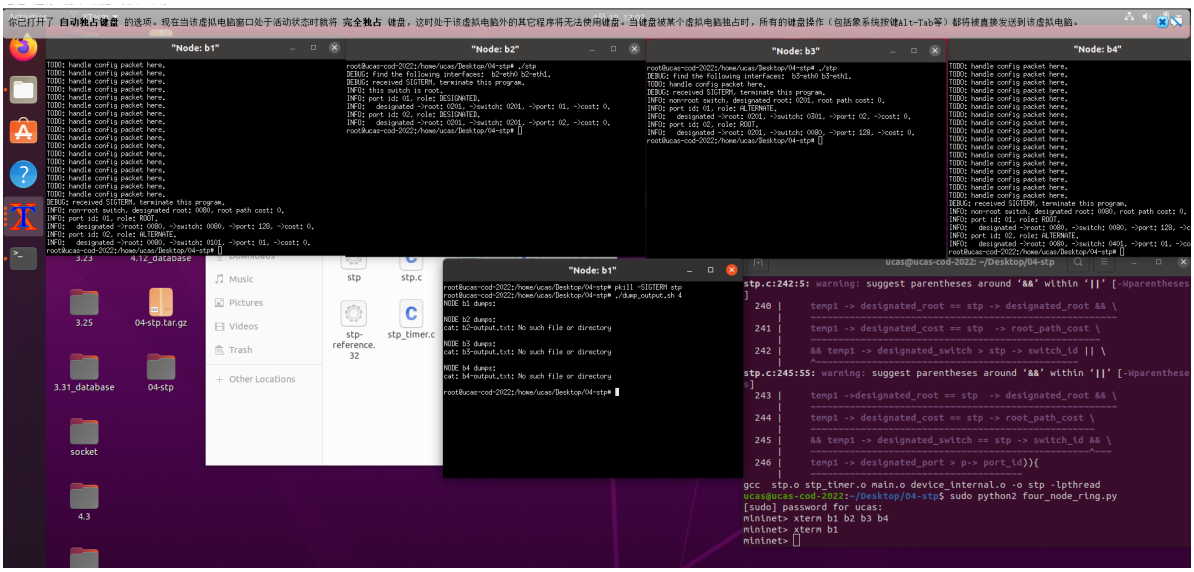
```
static void stp_handle_config_packet(stp_t *stp,stp_port_t *p,
    struct stp_config *config)
{
    // TODO:handle config packet here
    fprintf(stdout,"TODO: handle config packet here.\n");

    /*
    1. 根结点ID小的一方优先级高
    2. 到根结点开销小的一方优先级高
    3. 发送Config消息的结点ID小的优先级高
    4. 发送Config消息的端口ID小的优先级高
    */
    if(p->stp != stp){
        return;
    }

    /*设置优先级，如果Config为1，则表明发送来的Config优先级高，否则本地的Config优先级高
*/
    int priority = 1;
    if(config->root_id > stp -> designated_root){
        priority = 0;
    }
    else if(config->root_id == stp -> designated_root && \
        config -> root_path_cost > stp -> root_path_cost){
        priority = 0;
    }
    else if(config->root_id == stp -> designated_root && \
        config -> root_path_cost == stp -> root_path_cost \
```

```
        && config -> switch_id > p->designated_switch){
        priority = 0;
    }
    else if(config->root_id == stp -> designated_root && \
        config -> root_path_cost == stp -> root_path_cost \
        && config -> switch_id == p->designated_switch && \
        config -> port_id > p-> designated_port){
        priority = 0;
    }

    if(priority == 1){
        /*收到的Config的优先级高
        得到Config就需要对本地存储的数值进行更新，p和stp都要修改*/
        stp -> designated_root = config -> root_id;
        /*如果上一跳结点不是自己，说明是同一网段发送的Config消息，则cost值不需要改变，
        否则上一跳结点是自己，说明是不同网段发送的消息，cost值需要改变*/
        if(config -> switch_id == stp){
            stp -> root_path_cost = config -> root_path_cost + 1;
        }
        else{
            stp -> root_path_cost = config -> root_path_cost;
        }
        /*stp + 1 表示是上一跳*/
        p -> designated_root = config -> root_id;
        p -> designated_switch = config -> switch_id;
        p -> designated_port = config -> port_id;

        if(config -> switch_id == stp){
            p -> designated_cost = config -> root_path_cost + 1;
        }
        else{
            p -> designated_cost = config -> root_path_cost;
        }
        /*更新结点状态*/
        /*确定根端口
        首先遍历所有的非指定端口，并且寻找优先级最高的端口，首先设置一个初始值
        如果遍历得到的新的端口优先级更高，则替换，从而找到优先级最高的*/
        stp_port_t *new_root_port = &stp->ports[0];
        for(int i = 0; i < stp->nports; i++){
            stp_port_t *temp = &stp ->ports[i];
            if(stp_port_state(temp)=="ALTERNATE" && (
                new_root_port->designated_root > temp -> designated_root ||
                new_root_port->designated_root == temp -> designated_root && \
                new_root_port -> designated_cost > temp -> designated_cost ||
                new_root_port -> designated_root == temp -> designated_root && \
                new_root_port -> designated_cost == temp -> designated_cost \
                && new_root_port -> designated_switch > temp->designated_switch
|| \
                new_root_port ->designated_root == temp -> designated_root && \
                new_root_port -> designated_cost == temp -> designated_cost \
                && new_root_port -> designated_switch == temp->designated_switch
&& \
                new_root_port -> designated_port > temp-> designated_port)
                )
            {
                new_root_port = temp;
            }
        }
```

```
            stp -> root_port = new_root_port;

            /*更新端口状态*/
            /*确定指定端口*/

            /*非指定端口更新为指定端口
            这里需要设定收到的Config值和旧值的比较，收到的Config值已经在之前更新了stp和p
            所以需要拿原来存的值和现在的值进行比较
            注意这时的指定结点和端口均要设为自己，因为需要保证在同一网段内比较*/
            for(int i = 0; i < stp->nports; i++){
                stp_port_t *temp1 = &stp ->ports[i];
                if(stp_port_state(temp1) !="ROOT" && (
                    temp1 ->designated_root > stp -> designated_root ||
                    temp1 ->designated_root == stp -> designated_root && \
                    temp1 -> designated_cost > stp -> root_path_cost ||
                    temp1 -> designated_root == stp -> designated_root && \
                    temp1 -> designated_cost == stp  -> root_path_cost \
                    && temp1 -> designated_switch > stp -> switch_id || \
                    temp1 ->designated_root == stp  -> designated_root && \
                    temp1 -> designated_cost == stp -> root_path_cost \
                    && temp1 -> designated_switch == stp -> switch_id && \
                    temp1 -> designated_port > p-> port_id)){

                    temp1 -> designated_root = stp->designated_root;
                    temp1 -> designated_cost = stp -> root_path_cost;
                    temp1 -> designated_switch = stp -> switch_id;
                    temp1 -> designated_port = p -> port_id;
                }
            }

        }
        else{
            /*否则Config消息优先级低，直接从p端口转发出去*/
            stp_port_send_config(p);
        }
    }
```

4.调试过程

上述实验代码得到了一些输出，但与期望的结果有一些误差。



首先注意到端口号出现问题：需要将config的字节序进行转换。

调试后的最终代码如下，具体改动标注：

```
static void stp_handle_config_packet(stp_t *stp,stp_port_t *p,
    struct stp_config *config)
{
    // 进行Config消息的转换，与发送的hton相反，采用ntoh

    config->root_id = ntohll(config->root_id);
    config->root_path_cost = ntohl(config->root_path_cost);
    config->switch_id = ntohll(config->switch_id);
    config->port_id = ntohs(config->port_id);

    if(p->stp != stp){
        return;
    }

    /* 这里由于传入的Config消息，首先需要考虑是否对收到Config消息的端口信息进行改变，
    如果优先级高于p端口，则p端口记录的信息需要改为Config信息
    这里比较是与端口p进行比较*/
    int priority = 1;
    if(config->root_id > p -> designated_root){
        priority = 0;
    }
    /*因为记录的是本网段的信息，所以不需要 config -> root_id + 1
    我们只需要了解本网段到根结点的最短花销*/
    else if(config->root_id == p -> designated_root && \
        config -> root_path_cost  > p -> designated_cost){
        priority = 0;
    }
    else if(config->root_id == p -> designated_root && \
        config -> root_path_cost == p -> designated_cost \
        && config -> switch_id > p->designated_switch){
        priority = 0;
    }
    else if(config->root_id == p -> designated_root && \
        config -> root_path_cost  == p -> designated_cost \
        && config -> switch_id == p->designated_switch && \
        config -> port_id > p-> designated_port){
        priority = 0;
    }

    if(priority == 1){
        /*对方优先级更高，所以此时先更新p结点，p结点存储本网段最高优先级的网段信息
        stp在最后更新，实际上，stp的路径花销：
        stp到根结点的最短距离 = 根端口到根结点的距离(因为 stp结点就是通过根端口与根结点
通信) + 1
        最后的 + 1 需要解释一下，因为根端口存储的是所在网段的最小距离，所以还缺少一跳的
计数*/
        p -> designated_root = config -> root_id;
        p -> designated_switch = config -> switch_id;
        p -> designated_port = config -> port_id;
        p -> designated_cost = config -> root_path_cost;
        /*更新结点p状态，记录的是所在的网段的信息*/

        /*确定根端口
```

```
        这里调试出了问题，因为原先设置为"ALTERNATE",所以原来的根端口就不参与比较了
        而实际上所有非指定端口比较的含义是： 根端口一定不是指定端口
        遍历寻找最小值*/
        stp_port_t *new_root_port = &stp->ports[0];
        for(int i = 0; i < stp->nports; i++){
            stp_port_t *temp = &stp ->ports[i];
            if(stp_port_state(temp)!="DESIGNATED" && (
                new_root_port->designated_root > temp -> designated_root ||
                new_root_port->designated_root == temp -> designated_root && \
                new_root_port -> designated_cost > temp -> designated_cost ||
                new_root_port -> designated_root == temp -> designated_root && \
                new_root_port -> designated_cost == temp -> designated_cost \
                && new_root_port -> designated_switch > temp->designated_switch
|| \
                new_root_port ->designated_root == temp -> designated_root && \
                new_root_port -> designated_cost == temp -> designated_cost \
                && new_root_port -> designated_switch == temp->designated_switch
&& \
                new_root_port -> designated_port > temp-> designated_port)
                )
            {
                new_root_port = temp;
            }
        }
        /*这里开始更新stp*/
        stp -> root_port = new_root_port;

        stp -> designated_root = stp -> root_port -> designated_root;
        stp -> root_path_cost = stp -> root_port -> designated_cost + 1;

        /*更新端口状态*/
        /*确定指定端口*/

        /*非指定端口更新为指定端口
        根据自己本地的 port_id 来确定
        因为要么是旧值，过去本网段里所有的最高优先级的Config
        要么是新值，新值通过选举出的根端口(上一个网段）的最高优先级，到本网段的最高优先级
正好相差一跳的跳数，所以是stp被更新后的值进行比较
        旧值的Switch是存储的，和自己本地的消息（是否这个端口在收到Config消息后，因为
        root或cost的改变，而提高优先级，所以本端口（就是自己）变成了本网段的指定端口
        相应的数据是本地值*/
        for(int i = 0; i < stp->nports; i++){
            stp_port_t *temp1 = &stp ->ports[i];
            if(stp_port_state(temp1) !="ROOT" && (
                temp1 ->designated_root > stp -> designated_root ||
                temp1 ->designated_root == stp -> designated_root && \
                temp1 -> designated_cost > stp -> root_path_cost ||
                temp1 -> designated_root == stp -> designated_root && \
                temp1 -> designated_cost == stp  -> root_path_cost \
                && temp1 -> designated_switch > stp -> switch_id || \
                temp1 ->designated_root == stp  -> designated_root && \
                temp1 -> designated_cost == stp -> root_path_cost \
                && temp1 -> designated_switch == stp -> switch_id && \
                temp1 -> designated_port > temp1-> port_id)){

                temp1 -> designated_root = stp->designated_root;
                temp1 -> designated_cost = stp -> root_path_cost;
                temp1 -> designated_switch = stp -> switch_id;
```

```
                temp1 -> designated_port = temp1 -> port_id;
            }
        }
    }
    else{
        /*发送Config消息*/
        stp_port_send_config(p);
    }
}
```

实际上也可以这么理解：

root_port: 连接到根结点的最短路径，一定优先级最高，换言之，设 root_path_cost = s,s 是一个最小值

stp-> root_path_cost = s + 1，上一网段 + 一跳花销

其余端口：root_path_cost < s, 否，成为根端口

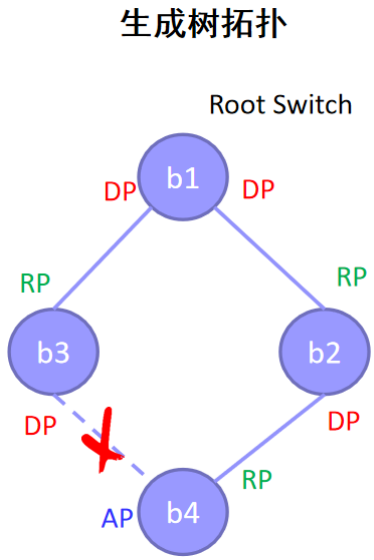
root_path_cost = s 可以

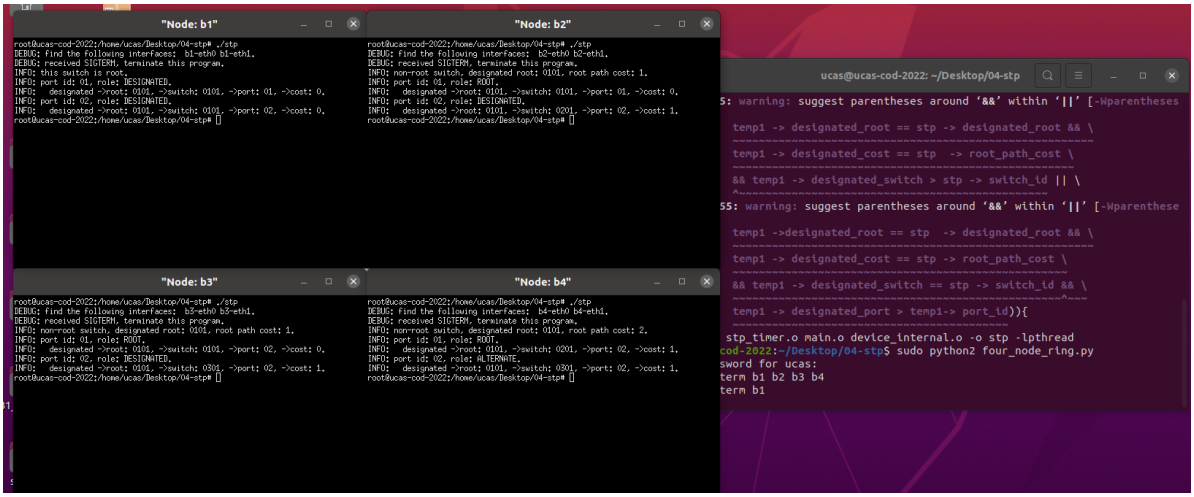
root_path_cost > s, 说明在本网段内，到根结点的路径花销至少为 s + 1, 则不如通过 stp 端达到，说明此时的指定端口为本网段的其它端口，但这种情况是不会长久成立的，很快通过比较会被更新为本网段的端口。

最终根据其余端口的三种情况分析，最终第一和第三种情况会消失，最终趋于稳定。

实验结果

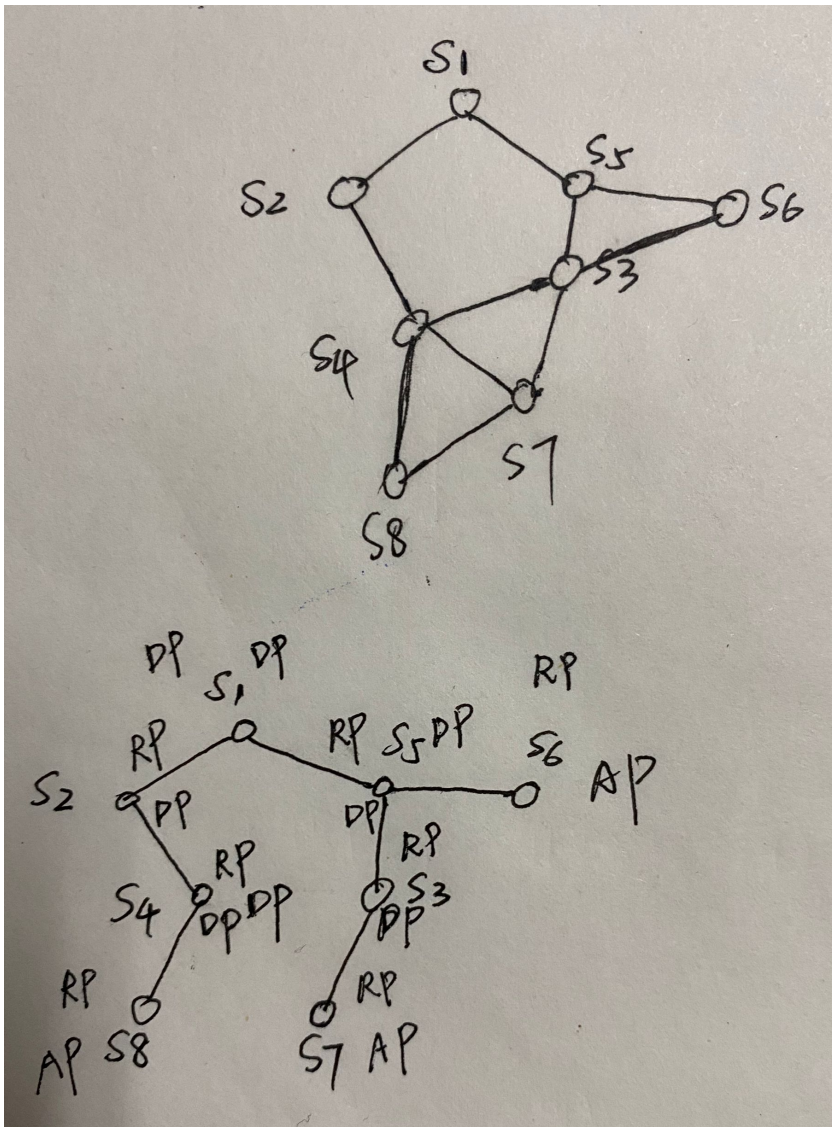
four_node_ring.py





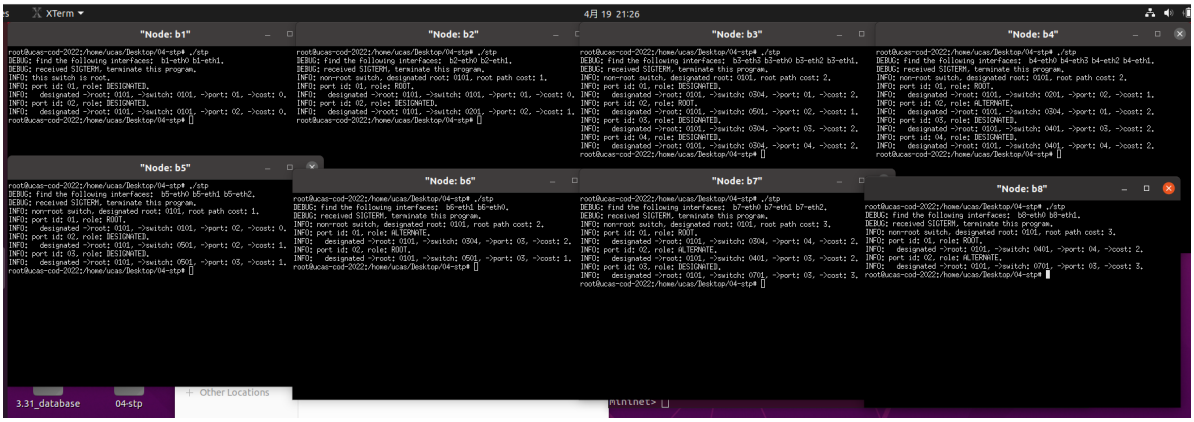
eight_node_ring.py

构造的拓扑如下:



运行 eight_node_ring.py文件

这里根据S4和S7结点的情况，找出了stp赋值上的一些错误，最终得到结果:



oj

日期	用户名	实验名称	作业状态	描述
2022-04-19 21:08	2018k8009909006	生成树实验	通过	stpHub_test:Pass; ring4_test:Pass; ring8_test:Pass;

调研思考题

1. 标准生成树协议中，如何处理网络拓扑变动的情况：当结点加入时？当结点离开时？

   解:设置老化时间，由于交换网络中建立交换网络学习表，当结点加入或离开时，网络的拓扑结构变动，导致某些表项在超时后自动被删除。建立新的MAC地址对应表项。根据TCN（Topology Change Notification)BPDU报文获得拓扑变动信息。此时需要用到本实验中的flag，设置为1表示拓扑结构发生变化。

2. 标准生成树协议是如何在构建生成树过程中保持网络连通的?

   解: 可以将多余的链路（非指定端口）阻塞，设置多个状态，其余端口可以执行转发功能。如果出现生成树中的链路断开，STP需要重新启用备用链路，将阻塞状态变为可转发的状态。

3. 快速生成树机制的原理?

   解：在STP上进行改进，删除了3种端口状态，新增加了2种端口角色。

   Alternate端口和Backup端口,前者学习到其它端口BPDU报文而阻塞的端口，是根端口的备份端口。Backup端口学习到自己发送的BPDU报文而阻塞的端口，是指定端口的备份端口。

   这两个端口只有Discard状态。根端口、指定端口都有forwarding、learning、discarding状态。

   在拓扑稳定后，无论非根桥是否收到根桥传来的BPDU报文，非根桥都在 hello timer 规定时间发送报文，完全由报文自主控制。如果一个端口连续 3 个 Hello Time 时间内没有收到上游设备发送过来的配置BPDU,那么该设备认为与此邻居之间的协商失败。

   检测拓扑变化的标准：一个非边缘端口迁移到 Forwarding 状态。

   一旦检测到拓扑发送变化，将进行如下处理：

   - 为本交换的所有非边缘指定端口启动一个 TC While Timer,时间为 hello time 的 2 倍，清空状态发生变化的端口上学习到的MAC地址。由这些端口向外发送 RST BPDU,其中TC置位。
   - 收到TC BPDU 的交换机清空所有端口的mac 地址，除了边缘端口和收到 STP BPDU 的交换机的端口。（正向从这个接口学习到 mac 地址，反向从这个接口的后面删除 mac 地址)
   - 继续泛洪 tcb pdu,实现全网mac地址表的更新。